Executive Summary **Late Days Left: 2**

Assignment Overview:

The assignment adds to Project 2; a multi-threaded key-value store system with communication facilitated through Java RMI (Remote Method Invocation), by achieving replication over 5 servers to ensure high availability. As the operations are performed concurrently, mutual exclusion needs to be handled at each server. The client should be able to call PUT and DELETE operations to any of the five servers and get consistent data using GET calls from any of the replicas. The PUT and DELETE operations should use 2-phase commit protocol to achieve consistency between updates. Acknowledgments can be used to inform the client about the status of the request. The scope of the project includes designing and implementing the server-side functionality to handle the above operations while ensuring mutual exclusion to prevent data inconsistency and replicating the server functionality across 5 different servers. The client must also pre-populate the key-value store with data, conduct 5 PUT, 5 GET, and 5 DELETE operations, and communicate with the server via RMI for these actions. Thus the assignment aims to demonstrate the implementation of distributed computing, multi-threading, and concurrent access control to build a robust, replication and concurrent key-value store using Java RMI.

Technical Impression:
I have learned the following during my implementation of the project:

The system consists of a coordinator and multiple participant servers. The coordinator is responsible for managing the distributed transaction. It has references to all the participant servers. Its key methods are:

- prepareTransaction() - Calls prepare() on each participant and collect their votes (true/false).
- commit() - If all participants voted true, calls commit() on each to make changes permanent.

Each participant server contains a map that will be changed by the transaction. It has a reference to the coordinator. Its key methods are:

- prepare() - Validates the transaction changes locally. Returns true if it's OK to commit, false otherwise.

- commit() - Makes the transaction changes permanent/visible. In this case, updates the map.

The flow works as follows:

- The client calls the required operations(PUT/GET/DELETE) on any of the 5 replicas(Participants).
- The Participant then calls the prepareTransaction() on coordinator.
- Coordinator calls prepare() on each participant
- Participants vote true/false
- If all participants return true then the coordinator calls commit() on each participant
- Each Participant makes changes permanent by updating it's map.
- The status is then returned to the client
    - GET: either the value or not key not found
    - PUT: if the key is already present then the operation is **aborted** otherwise, key is added(success).
    - DELETE: If the key is not present then the operation is **aborted** else deleted(success).

The rollback functionality is not implemented.

The synchronized keyword is used on prepare and commit methods to handle thread synchronization and avoid critical section problems. More fine-tuned locking can be achieved using re-entrant read-write locks.

Adding on to previous projects, the logger also includes each server by uniquely identifying it with its port.

If 15000 is the given port parameter then the following ports will be used.

- Coordinator: 15000
- Participant 1: 15001
- Participant 2: 15002
- Participant 3: 15003
- Participant 4: 15004
- Participant 5: 15005

````
# key-value-store


### Executable


```

keystore.jar

```


### Program Arguments


```

-server    -> run as server

-client    -> run as client

-port      -> server rmi port

```


### How to run server?


```

java -jar keystore.jar -server                    -> defaults to port 1099

java -jar keystore.jar -server -port 15000

```


### How to run client?


```
````
````

```
java -jar keystore.jar -client                    -> defaults to server rmi port
1099

java -jar keystore.jar -client -port 15000

```


Log files location


```

user home dir

%h/client%u.log     -> client logs

%h/server%u.log     -> server logs

```


Commands


```

post <key> <value>

put <key> <value>

get <key>

delete <key>

Application also pre-populates 10 post entries, 5 put entries, 5 gets and 5
deletes from key_value.txt file

```